SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 421–431 (1996)

Practice

# A Software Tool for Maintaining File and Macro Build Dependencies

JIM BUFFENBARGER

*Department of Mathematics, Boise State University, 2000 University Drive, Boise, Idaho 83725, U.S.A. and Hewlett-Packard Company, Mail Stop 207, 11311 Chinden Blvd., Boise, Idaho 83714, U.S.A.*

SUMMARY

**Translating each of a UNIX software system's source files into an object file, and linking the object files into an executable file, can be a time-consuming process. Object file generation accounts for most of this time. Tools that recognize and avoid unnecessary object file regeneration are very popular. Many such tools are named MAKE.**

**A software system often contains a parameter file. A parameter file is a source file, textually included by every other source file, which defines macros shared by multiple source files. Each object file depends on the parameter file, according to file dependencies. MAKE-like tools only understand file dependencies. So, when a software developer changes a macro in a parameter file, every object file is regenerated, regardless of whether its corresponding source file actually refers to the changed macro.**

**Unnecessary object file regeneration can be avoided by replacing some file dependencies with macro dependencies, which express that an object file depends on a macro.**

**DEP is a tool implementing a method for automatically detecting and maintaining a software system's file and macro dependencies. DEP works with any flavour of MAKE to minimize the cost of rebuilding an executable file.**

KEY WORDS: software builds; software dependencies; software construction; build avoidance; macro dependencies

## 1. INTRODUCTION

A software system typically contains a set of source files, which are assembled or compiled into a set of object files, which are linked into a set of executable files. A *build* is the process of generating the executable files. A *complete build* can be very costly, sometimes requiring hours of computing time. Therefore, methods and tools for performing incremental builds are popular.

An incremental build recognizes when a source file does not need to be reassembled or recompiled, because an existing object file is sufficient. This decision is typically made by some flavour of the popular software maintenance tool MAKE (Feldman, 1979). MAKE bases its decision on the following information:

- a representation of the dependencies between files, which arise due to file inclusion and file translation;

CCC 1040–550X/96/060421–11

*Received 11 June 1996*
*Revised 1 September 1996*

- an operating system's time-of-last-modification (TOLM) data, which are maintained for each file.

This paper introduces a method, and describes a production-quality tool, that combines file orientated dependencies with finer-grained dependencies. The finer-grained dependencies are those between a preprocessor macro's definition and files that refer to the macro's name. The software-maintenance tool is named DEP. DEP is part of the build process used by the Controller Section of Hewlett-Packard's Disk Memory Division.

Performing incremental builds according to file dependencies *and* macro dependencies has at least two advantages. The first benefit is economy. Finer-grained analysis prevents unnecessary retranslation, thereby conserving computing resources. The second benefit is more subtle, but just as real. Time is precious to a software maintainer; he or she will quickly recognize unnecessary retranslation and subvert a build process to avoid it. These *ad hoc* optimizations often compromise the reliability and robustness of a build process. Thus, finer-grained analysis indirectly preserves the correctness of a build.

The method described here applies to many software development environments. Admittedly, preprocessor macros are most often found in C and C++ programs, but they can be utilized in other programming languages as well. For example, assembly language 'equate' directives are effectively macro definitions. In addition, any programming language can be preprocessed by a tool like the UNIX macro processor M4 (Hewlett–Packard, 1992). For example, Icon (Griswold and Griswold, 1990) supports this. Nevertheless, the examples presented here are cast in the overwhelmingly popular language of C.

The reader is also expected to be familiar with UNIX, MAKE and Backus–Naur Form (BNF) (Aho, Sethi and Ullman, 1986).

Section 2 specifies the problem to be solved; Section 3 considers several possible solutions; Section 4 describes the chosen solution; Section 5 provides a brief example of the solution in action; and Section 6 briefly discusses experiences and draws conclusions.

## 2. THE PROBLEM

Software is developed as a set of modules. Each module is typically partitioned into an interface and an implementation, which are stored in separate files. For example, a module *f* might be stored in interface and implementation files f.h and f.c, respectively.

A module *f* *depends* on file g.h if a change to g.h requires *f* to be retranslated. Maintenance of these *file dependencies* is supported by tools like MAKE. File dependencies can be detected, and necessary retranslation can be performed, automatically.

A module rarely depends on all of another module's interfaces. For example, module *f* may only refer to some of the identifiers defined in file g.h. Yet, any change to g.h would cause *f* to be retranslated.

In fact, a software system often contains one file (or a small set of files) upon which every module depends. When such a file defines system-wide parameters, it is called a *parameter file* (Fowler, 1985). Parameter definitions often take the form of preprocessor-macro definitions (e.g., #define directives). A parameter file typically contains tens or hundreds of macro definitions, and is often textually included (via a #include directive) by every module in the system. Thus, a change to any macro's definition causes every module to be retranslated, even though a particular module may actually only depend on

a few definitions. Tools like MAKE, which support only file dependencies, force a module to effectively depend on every definition in the parameter file.

For example, suppose the macro definitions in Figure 1 are part of a parameter file named macs.h for a (low-quality) compiler. A symbol-table module probably contains

```
#include "macs.h"
```

but might depend only on the definition of MaxIdLen. Yet, a change to the definition of MaxNumLenLeft would retranslate the symbol-table module.

Maintenance of these *macro dependencies* is not supported by current tools. This paper presents a method for automatically detecting and recording macro dependencies for parameter files, and automatically performing necessary retranslation. DEP is a new tool, which implements the method. The method and tool are really only appropriate for environments employing parameter files.

Parameter files are a shining example of poor modularity. They are intended to be central and global. However, they are a fact of life in many software development and maintenance environments. Parameter files are sometimes generated automatically. For example, the environment in which DEP was developed generates the parameter file from a database of customer information.

The primary benefit of parameter files is that they centralize the control of build variability. A parameter file, in conjunction with conditional translation (e.g., #if directives), essentially selects one of many variants of the software for translation. Indeed, this technique is a legitimate alternative to the branching and merging operations of a software configuration management system (Tichy, 1985; Atria, 1992a).

The savings realized by building according to macro dependencies are multiplied in an environment where a build constructs every variant. For example, the environment in which DEP was developed builds one variant for each customer. Starting the construction of each variant in an empty directory is very wasteful of resources. File dependencies alone are just as wasteful, since every module includes the parameter file. Macro dependencies minimize the retranslation effort.

Macro dependencies are not the only kind of non-file dependency. For example, module *f* may only refer to some of the functions declared in g.h. Macro dependencies are the only kind of non-file dependency considered here.

## 3. ALTERNATIVES CONSIDERED

This section discusses several alternatives that were considered as candidate solutions to the macro-dependency problem, before the DEP solution was chosen. The DEP solution is described in Section 4.

```
/********** Max identifier length **********/
#define MaxIdLen 8
/********** Max number length, left of dot **********/
#define MaxNumLenLeft 8
/********** Max number length, right of dot **********/
#define MaxNumLenRight 4
/********** Max number length, plus sign and dot **********/
#define MaxNumLen (1 + MaxNumLenLeft + 1 + MaxNumLenRight)
```

*Figure 1. An example parameter file for a compiler*

A first alternative is to ignore macro dependencies and rely exclusively on file dependencies. This alternative is chosen by most software development projects. Its advantage is simplicity. The disadvantage of this coarse-grained view of dependencies, of course, is unnecessary retranslation.

A second alternative is to partition a parameter file into multiple files, and rely on file dependencies for build avoidance. This trick leads to a proliferation of little files. At worst, each macro definition needs its own file. It also decentralizes the control of build variability.

A third alternative is to use a flavour of MAKE that understands file dependencies and macro dependencies. Unfortunately, no such MAKE exists. NMAKE (Fowler, 1985) comes closest. At first, it seems to be a promising possibility.

*'The main features of NMAKE are that it: ... refines the granularity from the file to the variable level ... provides optional dynamic-dependency generation.'*

Its relevant features and deficiencies are described below.

NMAKE supports 'state variables', which can correspond to a #define directive in a file. However, NMAKE'S support is insufficient.

- Although NMAKE can generate file dependencies automatically, it cannot recognize and generate macro dependencies. A programmer must determine and maintain macro dependencies manually. This is a tedious and error-prone task, and the cost of a mistake can be high.
- NMAKE can only generate file dependencies from #include directives. Syntactically different, but semantically equivalent, directives in other languages (e.g., assembly language) are not recognized (Fowler, 1990).
- Even if a programmer is willing to maintain a parameter file's macro dependencies manually, NMAKE automatically generates a file dependency for the parameter file, effectively subsuming the macro dependencies. The .PARAMETER 'special rule' (or perhaps the -h command-line option) should prevent this behaviour, but it does not seem to.

A fourth alternative is to develop a set of shell scripts, which work with MAKE. They are effectively a prototype of the DEP solution. In order to recognize and process file dependencies and macro dependencies, they must perform several tasks.

- File dependencies must be computed and stored. There are several ways to do this. For example:
  - The GNU compiler's -MMD command-line option produces a file containing file dependencies, in makefile format (Free Software Foundation, 1995a).
  - The mkmf (Hewlett-Packard, 1992) program produces a makefile for a set of source files.
  - An AWK (Hewlett-Packard, 1992) script can search source files for #include directives.
- Macro dependencies must be computed and stored. This task can also be accomplished by an AWK script, which searches source files for occurrences of a known set of macro names.

- A parameter file must be recognized as such. This simply means that a parameter file's name must be removed from the previously computed file dependencies.
- Changes to files must be recognized. MAKE already does this.
- Changes to macros must be recognized. The current version of a parameter file can be compared with an older version. This task can be simple or difficult, depending upon the complexity of macro definitions. If macro definitions do not refer to other macros, changes can be recognized by a textual comparison (e.g., with an AWK script). Such is the case when definitions are just literals. In general, however, both versions of a parameter file must be preprocessed before macro definitions can be compared. Such is the case for the file in Figure 1.
- Retranslation due to file dependencies and changed files must be performed. MAKE already does this.
- Retranslation due to macro dependencies and changed macros must be performed. This task can be accomplished by deleting object files whose corresponding source files depend on changed macros. MAKE will then rebuild the deleted files.

## 4. THE DEP SOLUTION

DEP and MAKE work together (any flavour of MAKE is acceptable). DEP automatically maintains a file called a *depfile*, which expresses a software system's file dependencies and macro dependencies. As always, MAKE reads a makefile. A user is not expected to edit a depfile. The makefile is expected to textually include the depfile. Thus, MAKE actually reads the makefile and the depfile. This arrangement is illustrated in Figure 2.

### 4.1. Depfile syntax

DEP reads and writes a depfile. Strictly speaking, the language of depfiles is a subset of the language of plain makefiles (i.e., DEP does not require special makefile features). However, DEP utilizes lines that are treated as comments by MAKE (i.e., lines that begin with a # character). An extended-BNF grammar for depfiles is shown in Figure 3. Lower case names are non-terminals. Upper case names are terminals.

Each non-terminal dep is terminated by a new line character. Terminals are separated by one or more white-space characters. A white-space character is a space or tab. A COMMENT terminal is a sequence of characters not containing a new line character (a 'blank' line is also a comment). A PATHNAME or MACRONAME terminal is a sequence of characters not containing a white-space or new line character.

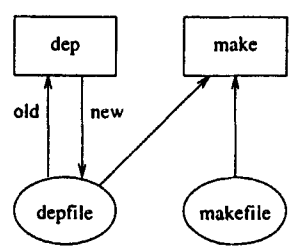


*Figure 2. Relationship of DEP and MAKE*

```
depfile     = dep*
dep         = file_dep
            | macro_dep
            | comment
file_dep    = file ':' files
macro_dep   = '#m' file ':' macros
comment     = '#c' COMMENT
            |
files       = file*
macros      = macro*
file        = PATHNAME
macro       = MACRONAME
```

*Figure 3. Depfile syntax*

## 4.2. Depfile semantics

The meaning of a depfile is similar to that of a makefile. Referring to the grammar in Figure 3, a file_dep line means exactly what it means in a makefile: the *target* file depends on zero or more *prerequisite* files. A macro_dep line means: the target file depends on every macro in macros, either directly or indirectly (via file inclusion). A comment line is ignored.

## 4.3. Overview of operation

DEP performs several functions, which are illustrated in Figure 4 and described below.

1. DEP begins by reading old dependencies from the depfile. The old dependencies were computed by the previous invocation of DEP.
2. DEP determines the set of targets whose dependencies need to be recomputed. This is actually the union of two sets, which are described below.
   - Targets without old dependencies: the build process creates an *inputs file*, which contains the names of targets whose dependencies must be computed for the first time. These are simply the names of targets that do not yet exist. For example, if foo.c is a source file and foo.o does not exist, then the dependencies of foo.o must be computed.
   - Targets with old dependencies, whose prerequisites have changed: for each target in the old dependencies, the target's TOLM is compared with those of its old prerequisites. This determines the names of targets whose dependencies must be recomputed.
3. DEP computes new dependencies. Initially, DEP recognizes built-in dependencies (e.g., foo.o depends on foo.cc or foo.c). Then, DEP reads source files, searching for the following.
   - File dependencies: a file dependency exists when a file-inclusion directive references a file name. A reference to a parameter file is appropriately ignored. Parameter-file names are provided by a command-line argument.
   - Macro dependencies: a macro dependency exists when a macro name is referenced. Macro names are contained in a *macros file*. More specifically, while scanning a source file, each lexeme matching an 'identifier' pattern is recognized, the set of reserved words for the programming language is consulted, and then the set of

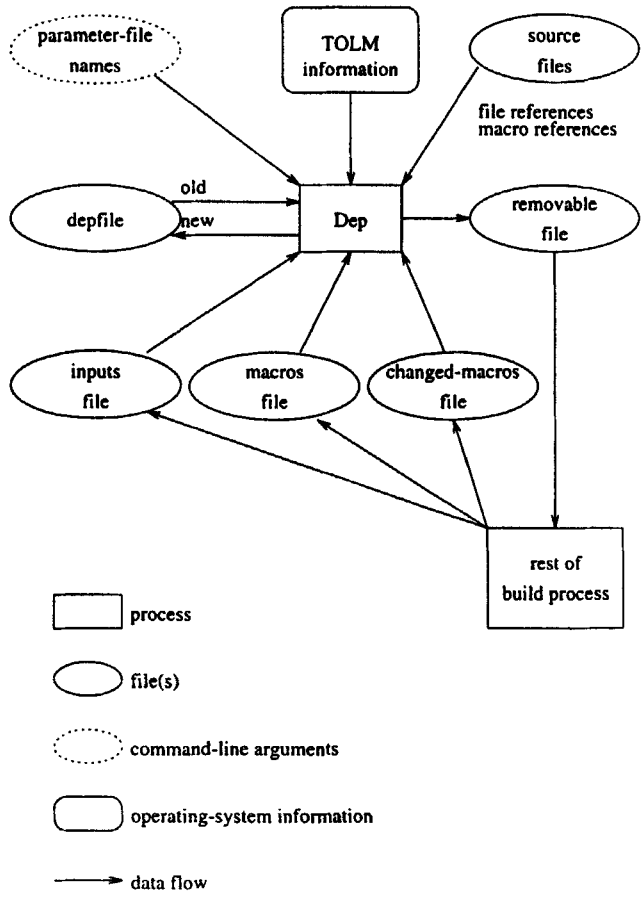


*Figure 4. Environment of DEP*

macro names is consulted. In general, a macro name cannot be recognized in source code without the set of all macro names.

4. DEP computes the names of targets that are inconsistent, according to macro dependencies and changed macros. The names of changed macros are contained in a *changed-macros file*. The names of inconsistent targets are written to a *removable file*. The build process is expected to delete them, forcing their reconstruction.
5. Finally, DEP overwrites the old depfile with the newly computed dependencies.

Perhaps surprisingly, DEP does not need to record a TOLM for each macro. Strictly speaking, DEP does not detect macro changes at all. Another part of the build process determines which macros have changed, and tells DEP. Macro changes can be detected by textually comparing a macro's previous and current definition.

## 4.4. Invocation

Figure 2 shows that DEP and MAKE work together. Several details of their collaboration remain to be discussed.

A programmer typically begins a build by invoking MAKE. MAKE reads the makefile and, due to textual inclusion, tries to read the depfile. For a first-time build, there may be no depfile, but that does not matter because there are no object files either. Then, the following events occur:

1. MAKE invokes DEP, to construct or reconstruct a depfile.
2. MAKE invokes DEP, to construct a removable file.
3. The files named in the removable file are deleted.
4. MAKE invokes itself recursively, rereading the makefile and reading the newly constructed depfile.
5. MAKE performs all necessary translation and retranslation, as usual.
6. The recursive invocation of MAKE terminates.
7. The original invocation of MAKE terminates.

Actually, steps 1 and 2 occur during a single invocation of DEP.

## 4.5. Implementation details

DEP is implemented in C and C++, using the GNU compiler (Free Software Foundation, 1995a), the GNU C++ library (Free Software Foundation, 1995b), FLEX (Paxson, 1990), and GPERF (Schmidt, 1989). FLEX is used to generate a scanner for C, C++ and INTEL i960 assembly language source files. GPERF is used to generate perfect hash functions, to test if a candidate macro name is just a reserved word.

# 5. A SMALL EXAMPLE

This section presents a small example, which is intended to demonstrate:

- the content of a depfile;
- how a makefile uses a depfile;
- how a makefile updates a depfile.

The example refers to a system of four source files, two object files and an executable file. The files and their dependencies are illustrated in Figure 5. A file-dependency arc

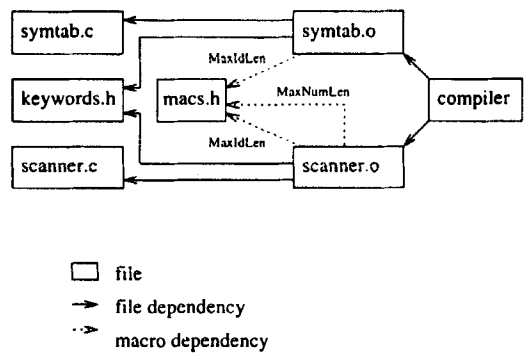


*Figure 5. Dependency graph of example system*

```
PROGRAM:=compiler
MODULES:=symtab scanner
SOURCES:=$(addsuffix .c,$(MODULES))
OBJECTS:=$(addsuffix .o,$(MODULES))

all: dependencies
      $(MAKE) $(PROGRAM)

$(PROGRAM) : $(OBJECTS)
      gcc -o $@ $^

%.o: %.c
      gcc -c $<

depfile:
      touch $@ macs.hh

include depfile

dependencies:
      echo $(foreach f,$(OBJECTS), \
          $(shell test -f $(f) || echo $(f))) > inputs
      awk -f compute-macros.awk < macs.h > macros
      awk -f compute-changed.awk macs.hh macs.h > changed
      dep -i inputs -M macros -m changed -u macs.h \
          | xargs rm
      cp macs.h macs.hh
      rm inputs macros changed
```

*Figure 6. Example makefile*

points at the prerequisite. A macro-dependency arc points at the parameter file, and is labelled with the macro's name.

Although DEP can be used with any flavour of MAKE, a GNU makefile (Stallman and McGrath, 1991) for the system is shown in Figure 6.

The rule for target depfile ensures that a depfile exists, allowing MAKE to read it. It also ensures that an 'old' version of macs.h exists, allowing macro definitions to be compared. The include depfile directive causes MAKE to read the depfile. The rule for target dependencies updates the depfile and deletes files that depend on changed macros. Target all depends on target dependencies; its command is a recursive invocation of MAKE, which causes MAKE to read the updated depfile.

A depfile for the system is shown Figure 7.

## 6. RESULTS AND CONCLUSIONS

Most software systems are built incrementally, according to file dependencies alone. We have seen how the common practice of gathering macro definitions into a parameter file

```
scanner.o : scanner.c keywords.h
#m scanner.o : MaxNumLen MaxIdLen

symtab.o : symtab.c keywords.h
#m symtab.o : MaxIdLen
```

*Figure 7. Example depfile*

introduces unnecessary file dependencies, which cause unnecessary retranslation. We have also seen how these unnecessary file dependencies can be replaced by macro dependencies, which avoid the unnecessary retranslation. The method is especially valuable in an environment where multiple variants of software are built, and a particular variant is selected by macro definitions in a parameter file.

Without this method, a software developer naturally hesitates to change a parameter file, knowing that the change will trigger a complete build. If a parameter file must be changed, a developer will attempt to avoid a time-consuming build by manually retranslating only what he or she thinks is necessary. This *ad hoc* attempt at optimization can easily produce inconsistent software.

This method can be implemented as a set of scripts or as a production-quality solution. Unfortunately, experience shows that script-based prototypes tend to have limitations or other bugs. For example, about two years prior to DEP'S introduction, a prototype similar to that discussed in Section 3 was developed, but it could only process nested textual inclusion to a certain depth (depending upon the maximum number of open files allowed by AWK). DEP is a production-quality program, which works with any MAKE, ensuring that builds are quick, reliable and consistent.

As mentioned earlier, DEP is part of the build process used by the Controller Section of Hewlett-Packard's Disk Memory Division. This process supports software development in assembly language, C and C++. The development platforms are UNIX and DOS. A complete build actually constructs tens of variants. The depfile for this environment has the following (approximate) characteristics:

- 60 000 characters,
- 1 200 lines,
- 327 source files,
- 229 object files,
- 34 macros,
- 1 805 file dependencies,
- 213 macro dependencies.

Notice that each source file depends upon an average of 0·65 macros, yet 168 source files textually include the entire parameter file.

For rough timing estimates, DEP was run, in the abovementioned environment, on a 40MHz/80486/DOS machine with a typical-speed disk. Approximately two minutes were required to construct a depfile from scratch. For a previously constructed depfile and no macro changes, approximately 15 seconds were required to determine that no updating was necessary. These times are simply intended to demonstrate that DEP is reasonably quick.

## Acknowledgements

I would like to thank my colleagues at HP and BSU for supporting this work. In addition, I greatly appreciate the comments from the *Journal's* anonymous reviewers.

## References

Aho, A., Sethi, R. and Ullman, J. (1986) *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.

Atria (1992a) *ClearCase Concepts Manual*, Atria, Inc., Natick, MA.
Atria (1992b) *ClearCase Reference Manual*, Atria, Inc., Natick, MA.
Cichinski, S. and Fowler, G. (1988) 'Product administration through SABLE and NMAKE ', *AT& T Technical Journal*, **67**(4), 59–70.
Clemm, G. (1988) 'The Odin specification language', in *Proceedings of the International Workshop on Software Version and Configuration Control*, ACM, Baltimore MD, pp. 144–158.
Erickson, V. and Pellegrin, J. (1984) 'Build: a software construction tool', *AT&T Bell Laboratories Technical Journal*, **63**(6), 1049–1059.
Feldman, S. (1979) 'Make: a program for maintaining computer programs', *Software: Practice and Experience*, **9**(4), 255–265.
Fowler, G. (1985) 'The fourth generation make', *USENIX Portland 1985 Summer Conference*, Portland, Oregon, USA, USENIX Association, Berkeley, CA, pp. 159–174.
Fowler, G. (1990) 'A case for make', *Software: Practice and Experience*, **20**(1), 35–46.
Free Software Foundation (1995a) *GNU Compiler*, anonymous ftp: prep.ai.mit.edu.
Free Software Foundation (1995b) *GNU Compiler*, anonymous ftp: prep.ai.mit.edu.
Griswold, R. and Griswold, M. (1990) *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ.
Haag, V. (1993) *MKS Toolkit: Make*, Mortice Kern Systems, Waterloo, Canada.
Hewlett-Packard (1992) *HP-UX Reference (Volume 1)*, Hewlett-Packard Company, Palo Alto, CA.
Hume, A. (1987) 'Mk: A successor to make', in *USENIX Phoenix 1987 Summer Conference*, Phoenix, Arizona, USA, USENIX Association, Berkeley, CA, pp. 445–457.
Paxson, V. (1990) *Using Flex*, anonymous ftp: prep.ai.mit.edu.
Schmidt, D. (1989) *Gperf Manual*, anonymous ftp: prep.ai.mit.edu.
Stallman, R. and McGrath, R. (1991) *GNU Make: A Program for Directing Recompilation*, anonymous ftp: prep.ai.mit.edu.
Tichy, W. (1985) 'RCS: a system for version control', *Software: Practice and Experience*, **15**(7), 637–654.
Walden, K. (1984) 'Automatic generation of make dependencies', *Software: Practice and Experience*, **14**(8), 575–585.

**Author's biography:**

**Jim Buffenbarger** is a Shared Professor in Computer Science for Hewlett-Packard and Boise State University. He joined both organizations in 1991. He earned a B.S. degree in computer science from California State University at Hayward, an M.S. degree in computer science from San Jose State University, and a Ph.D. in computer science from the University of California at Davis.